Design Ideas for Markov Chain Monte Carlo Software

Todd L. Graves

July 28, 2006

Keywords: Java, Metropolis–Hastings algorithm, updating schemes, expressing statistical models

Abstract

This article discusses design ideas useful in the development of Markov chain Monte Carlo (MCMC) software. Goals of the design are to facilitate analysis of as many statistical models as possible, and to enable users to experiment with different MCMC algorithms as a research tool. These ideas have been used in YADAS, a system written in the Java language, but are also applicable in other object-oriented languages.

1 Introduction

Bayesian analysis is now a practical option for most statistical problems, and most of these require Markov chain Monte Carlo algorithms (MCMC). Tools and ideas that assist researchers in quickly developing MCMC software are of great importance: indeed, in his Journal of the American Statistical Association vignette about Bayesian analysis, Berger (2000) notes that "creation [of 'automatic' Bayesian software] should... be a high priority for the profession." In this paper I discuss lessons I have learned about design of MCMC software while developing YADAS (Graves, 2003). Goals of the software design discussed here include extensibility, both with respect to new models and new types of MCMC algorithms, and code reuse so that common pieces of analyses can be easily replicated. (Berger (2000) points out that the large class of problems amenable to Bayesian analysis and the recent, continuing expansion of the class of available computational techniques are two factors militating against the development of general Bayesian computational software, although they are both major strengths of the paradigm.) More specific goals include making it easy to make small changes to analyses already performed, including to prior parameters, but also distributional forms, link functions, and so forth. The design aims to give the user the option of either ignoring the details of algorithm implementation and obtaining adequate performance in many problems, or of creating arbitrary new algorithms when more careful tailoring is productive. The design utilizes principles of object-oriented

design that formalize the commonalities between similar operations and facilitate extensions that follow these commonalities.

The remainder of this section gives further motivation for the design based on the great generality of the family of Metropolis-Hastings algorithms, followed by introduction of terminology, especially that used in object–oriented programming. Section 2 contains definitions of the key components of the software design. Section 3 features two example problems that illustrate the design in action. Section 4 discusses related software efforts. An appendix includes an example YADAS code listing and discusses how to learn more about YADAS.

1.1 The Metropolis–Hastings algorithm

The generality and easy applicability of the Metropolis–Hastings algorithm motivates many of the design decisions made here. Suppose that one wishes to sample from the unnormalized posterior density $\pi(\theta)$ for $\theta \in \Theta$. For $\theta \in \Theta$, let $T(\cdot|\theta)$ be a probability density function on Θ. The Metropolis–Hastings algorithm can be defined as follows. Let $\theta^0 \in \Theta$ be the initial value. Then, for $k = 1, \ldots, N$,

1. *Propose* a new value θ' by sampling from the density $T(\cdot|\theta^{k-1})$.
2. *Accept* the new value with probability

$$\min\left\{1, \frac{\pi(\theta')}{\pi(\theta^{k-1})}\frac{T(\theta^{k-1}|\theta')}{T(\theta'|\theta^{k-1})}\right\}.$$

   If the new value is accepted, set $\theta^k = \theta'$, otherwise set $\theta^k = \theta^{k-1}$.
3. Return to Step 1.

This algorithm can be implemented in great generality provided that it is possible to

1. Calculate ratios $\pi(\theta')/\pi(\theta)$;
2. Sample from the transition density $T(\cdot|\theta)$;
3. Calculate ratios $T(\theta|\theta')/T(\theta'|\theta)$. For some special cases this ratio is always 1, in which case the algorithm is called the Metropolis algorithm.

Furthermore, one can construct algorithms based on several choices of $T_j(\cdot|\theta)(j = 1, \ldots, J)$ by cycling through proposals based on $T_1, \ldots, T_J$. An important example is *variable-at-a-time* Metropolis: if $\theta = (\theta_1, \ldots, \theta_J)$

with each $\theta_j \in R$, one can let

$$T_j(\theta'|\theta) = (2\pi s_j)^{-1/2} \exp\{-(\theta'_j - \theta_j)^2/(2s_j^2)\} I\{\theta'_i = \theta_i, i \neq j\},$$

i.e., T_j lets $\theta'_j \sim N(\theta_j, s_j^2)$. Under easily achievable conditions, algorithms based on these steps will produce samples from distributions that converge to π. See Chib and Greenberg (1995), and Tierney (1994). Metropolis *et al* (1953) and Hastings (1970) are important historical papers about the Metropolis algorithm. As an example of the generality of these algorithms, Gibbs sampling is a special case: if one lets $T_j(\theta'|\theta) = \frac{\pi(\theta_1,\ldots,\theta_{j-1},\theta'_j,\theta_{j+1},\ldots,\theta_J)}{\int \pi(\theta_1,\ldots,\theta_{j-1},\lambda,\theta_{j+1},\ldots,\theta_J)d\lambda} I\{\theta'_i = \theta_i, i \neq j\}$ be defined by the conditional distribution of θ_j given $\theta_1, \ldots, \theta_{j-1}, \theta_{j+1}, \ldots, \theta_J$, then the acceptance probability is always 1. Most often it is impractical to sample from the conditional distributions or even to evaluate them. Even when it is possible, there are costs associated with coupling the proposal distributions T_j with the posterior distribution π: if the model changes, the code to analyze it will change, whereas more versatile choices of T_j (perhaps with some tuning done behind the scenes) imply software that is easier to change. The Metropolis–Hastings framework also includes reversible jump MCMC (Green, 1995), and versions of data augmentation (Tanner and Wong, 1987), simulated tempering (Geyer and Thompson, 1995), and other algorithms. This approach is powerful in part because it generalizes readily to simultaneous updates of multiple correlated parameters.

In large part, the purpose of the software design described here is to help the user define π and appropriate choices of T_j.

1.2 Conventions and terminology

When a concept needs a name, this article will use the names used in YADAS, except where noted. As much as possible, when discussing the software design, I use the terminology of Java to be concrete. For example, the term *array* refers to a one–dimensional, fixed–length structure for storing multiple values of the same type. Two-dimensional arrays are arrays of arrays, with no implication that the inner arrays are of the same length (the outer array can be ragged). Array indexing starts at zero, unlike in S.

This design utilizes object-oriented ideas: it has been implemented in Java, but the ideas can also be applied in other object-oriented environments.

Object-oriented programming consists of manipulations of *objects*. The abstract description of like objects is referred to as a *class*; a specific object used in a program that is an example of a class is known as an *instance* of that class. (A class is a sort of template, and instances are things that follow that template.) Classes are like data structures in non-object-oriented programming in that they can contain several pieces of data of different forms, but they also can contain *methods*, a particular type of procedure or function that

is associated with a specific class. A method may return data of some type or an object, or it may not, in which case its return type is said to be void. A special type of method is a *constructor*: its behavior is to create a new instance of the class. In Java and in this paper, the syntax for calling a method `f` on an object `x` is `x.f()`, possibly with arguments inside the parentheses. Sometimes `f` can be called with different numbers of different types of arguments; these different argument possibilities are called *signatures* of the method `f`. Object-oriented systems also have class hierarchies: if a class is built based on an existing class to contain all the same functionality, perhaps with some of it reimplemented, and perhaps with some additional functionality, it is known as a *subclass* or *derived class* of the first class (the *superclass* or *base class*) or is said to *inherit* from that class; one often sees extensive diagrams of inheritance structure for large systems. Inheritance is a powerful way to reuse code, as new classes that are similar to old classes can be implemented with only the differences requiring new code. The class hierarchy described in the present paper is actually fairly simple; most of the structure is represented using *interfaces* (this is the Java term; the term *protocol* is more generally used in other object-oriented contexts). Broadly speaking, an interface is a definition of how objects conforming to that interface can be interacted with; more concretely, an interface is a collection of names of methods, and a class is said to *implement* the interface if it contains definitions for all those methods. (Some object-oriented languages, including C++, define interfaces using *abstract classes* and *pure virtual functions*.) In Java, each class inherits from exactly one superclass, but it may implement arbitrarily many interfaces. The interface leads to one of the most important object-oriented programming techniques: one can construct an array or other collection of objects that implement the same interface, and then loop over them, calling methods that belong to the interface, which can lead to somewhat different appropriate behavior for each object. The key concepts of the design described here are predominantly interfaces, including the interface `MCMCBond`, which defines terms in posterior distributions, and `MCMCUpdate`, which defines steps in MCMC algorithms.

Many good introductory books on Java and object-oriented programming exist; a favorite is Eckel (1998).

2 Design ideas

This section presents the key elements of the proposed software design. They fall in five categories: inputs (such as data), entities whose posterior distribution is desired, terms in the posterior distribution, steps in the MCMC algorithm, and outputs. Each of these is organized by an interface.

Interface definitions here use Java syntax. A generic interface definition is:

```
interface interface-name {
   return-type method-name (arg1-type arg1-name, arg2-type arg2-name, ...);
```

```
}
```

where `return-type`, `arg1-type`, and so on can be, for example, `double` for a real number, `int[]` for an array of integers, `String[][]` for an array of arrays of strings, or `void` for no return value. There can be zero or more arguments. For example, the interface `LogDensity`, discussed in §2.3.1, consists of the single method `compute`, which accepts a two-dimensional array (matrix) of inputs and returns a real number:

```
interface LogDensity {
    double compute (double[][] args);
}
```

I will colloquially refer to an object implementing, for example, the `LogDensity` interface as "a `LogDensity`".

2.1 Data and other inputs

The interface `MCMCInput` is used to define inputs to the MCMC analysis, including data and covariates, constants such as fixed prior parameters, and initial values for parameters. The two key methods are `r` and `i` for handling real- and integer-valued inputs respectively:

```
interface MCMCInput {
    public double[] r(String name);
    public int[] i(String name);
}
```

(If reals and integers can be handled together, as in S, only one of these is needed.) An example of a class that implements the interface is `DataFrame`, intended to be similar to the class of the same name in S. For example, if `d` is a DataFrame, `d.r("y")` returns an array of real numbers associated with the name "y". `DataFrame` objects read in the values of all their variables from files, and all variables have same length but may have different types. `ScalarFrame` is another class implementing `MCMCInput` but is restricted to scalar variables; it exists to improve readability of input files. Potentially, new classes could handle streaming data.

The `MCMCInput` architecture helps write analysis code that can be reused with different data sets: only the `MCMCInput` objects need to be changed.

2.2 The `MCMCNode` Interface

The MCMC algorithm's purpose is to sample from the joint posterior distribution of a number of entities. These entities will often be represented as nodes in a graphical model, and in this design are stored in objects that implement the `MCMCNode` interface. The most obvious example of an `MCMCNode` is an unknown parameter possibly of more than one element, but nodes may also include augmented data, parameter expansions, or control variables such as temperature in simulated tempering. `MCMCNode` objects may also contain data that are never updated: this may be useful for many reasons, such as anticipating another analysis where something that is now constant will become unknown, or it may be convenient to store observed and missing, imputed data together. The interface definition is

```
interface MCMCNode {
    double[] getValue ();
    void setValue (double[] newvalue);
    void addBond (MCMCBond bond, int which);
    MCMCBond[] relevantBonds ();
}
```

The method `getValue()` returns the value of the node at the current stage of the algorithm, while the method `setValue()` changes the value of the node to a new value (it may be useful to also have a method that changes a single element of the value). The method `setValue()` is called when an update step in the algorithm decides to change the node value. Another important behavior of a node is to help the software compute the unnormalized posterior distribution: to do this, it must communicate which terms of the posterior contain it. The `MCMCMBond` interface will be discussed later: it defines terms in the posterior distribution, which are relationships between nodes. In the `MCMCNode` interface, the `addBond` method is called when a new posterior term is defined, and each node in the bond makes a note that it belongs to that bond, and also which role it plays in the bond (encoded in the integer `which`, an argument to the `addBond` method). For example, if three parameters y, μ, and σ are related by $y \sim N(\mu, \sigma^2)$, the integer `which` may be 0 for the data parameter y, 1 for the mean parameter μ, and 2 for the standard deviation parameter σ. Later, while the algorithm is running, the method `relevantBonds()` can be called, it will return the array of bonds that feature the node, and the algorithm can calculate the change in posterior density that would result from changing the value of the node.

In YADAS, the main example of an `MCMCNode` is the `MCMCParameter` class, and it makes no distinction between unknown parameters, augmented or imputed data. An `MCMCParameter` can be multivariate if desired; for instance in an analysis of variance problem with group-dependent means, all the group means can be stored in the same parameter. More generally, several regression parameters can be stored together.

The `MCMCParameter` class has much more functionality than is required by virtue of being a node: in particular, it implements the `MCMCUpdate` interface (a parameter "knows how to update itself"): see §2.4.

2.3 Bonds: terms in the posterior distribution

The statistical model (unnormalized posterior distribution) to be analyzed is defined to be a sum of terms ("bonds") implementing the `MCMCBond` interface (if, as in YADAS, posterior computations are done on the log scale; otherwise it can be thought of as a product). Conceptually, to compute the unnormalized posterior distribution at a given value of the nodes, one loops over the bonds, computing each one, and adding the results. (The term "bond" arises from a chemistry interpretation of a graphical model, where each parameter is identified with an atom.) The interface definition for `MCMCBond` is as follows.

```
interface MCMCBond {
    double compute (several signatures);
    Collection getNodeList ();
}
```

Here `Collection` refers to a generic container that holds multiple objects. The method `getNodeList()` returns a list of the nodes required to define the bond.

The `compute()` method(s) can be defined and implemented in different ways depending on developer choice. There are a number of ways of thinking about how bonds compute themselves. Most obviously, a term in the posterior distribution is a function b of the nodes θ, so the `compute()` method could take θ as its argument and return $b(\theta)$. The YADAS approach is that b is most often needed in Metropolis-Hastings acceptance probability calculations in which one requires $\log\{b(\theta')\} - \log\{b(\theta)\}$, where θ is the current value and θ' is some perturbation. (When one requires the posterior density itself rather than ratios, this approach is a bit unnatural.) For this reason the `compute` method has several signatures corresponding to different ways of describing the perturbation. For example, if a bond depends on two parameters, $\mu = (\mu_i : 0 \le i < 5)$ and σ, and if the perturbation under consideration changes μ_2 to μ_2', the arguments to `compute()` must communicate that the second element of the zeroth parameter (μ) is the one potentially changing to μ_2'. One important capability that should be preserved for efficiency reasons, is that the bond should store its current value $b(\theta)$ rather than having to recalculate it.

A goal of this design is to make it possible to handle any unnormalized posterior distribution, and as described, the `MCMCBond` interface has not yet ruled this out: any unnormalized posterior distribution could be calculated inside a single bond. However, it also does not contain any support for the types of bonds that are commonly found in statistical analyses. It is the responsibility of the classes that implement the

MCMCBond interface to make it easier for users to describe commonly occurring bonds, as well as novel bonds with some commonly occurring aspects. In this design, one class that implements the MCMCBond interface, and thus far accounts for nearly all bonds, is the BasicMCMCBond class. This class is very versatile and is based on the observation that there are only a few different commonly used probability density functions: Gaussian, Gamma, Poisson, Binomial, Beta, StudentT, and so forth, and multivariate versions of some of these. The richness of the family of potential data analyses comes from the variety of ways of selecting arguments for these density functions. A BasicMCMCBond, then, consists of an array of nodes, an array of objects implementing the MCMCArgument interface, which are essentially functions that convert nodes into arguments for a density function, and a LogDensity, which takes the output of the MCMCArgument objects, runs it through the log density function, and returns the scalar value of the bond.

2.3.1 The LogDensity Interface

LogDensity is an interface (actually called Likelihood in YADAS), consisting of a single method,

```
interface LogDensity {
    double compute (double[][] args);
}
```

LogDensity objects are simply functions that calculate logs of probability density (or mass) functions. They are best understood through an example: the Gaussian class, which implements the LogDensity interface, expects as the contents of its args an array of data $y_i, 0 \leq i < n$, an array of means $\mu_i, 0 \leq i < n$, and an array of standard deviations $\sigma_i, 0 \leq i < n$, and its compute() method calculates

$$\sum_{i=0}^{n-1} \{-\log \sigma_i - (y_i - \mu_i)^2 / 2\sigma_i^2\}.$$

Note that this description handles several cases: the μ_i could all be different, could all be equal to a common μ, or anything in between; σ_i could be a constant, all equal to σ, or equal to $\lambda|\mu_i|$ for some known or unknown parameter $\lambda > 0$; and the y_i could be data, unknown parameters, or a mixture (i.e., some observed data and some missing data); μ_i and y_{i-1} could be the same in a random walk model; μ_i could be $\sum_j x_{ij}\beta_j$ for some regression coefficients β_j, and there are many other possibilities. (A cost of this versatility is a bit of computational inefficiency; the $-\log \sigma_i$ term is computed even if σ_i is not being updated, in which case most Bayesian statisticians would think of this term as a constant.) Because of this versatility, one can do many analyses with only a small number of LogDensity objects. Also note that if an analyst wants to change a distributional form from, say, Gaussian to StudentT, it is conceptually not much more difficult than changing the word Gaussian to StudentT in the code (except that the analyst must also define an

array of degrees of freedom parameters for the t distribution). Creating all the possibilities as arguments to the `LogDensity` object is the responsibility of classes implementing the `MCMCArgument` interface.

2.3.2 The `MCMCArgument` Interface

The `MCMCArgument` interface, actually called `ArgumentMaker` in YADAS, is the source of this design's ability to define relatively easily statistical models made up of common pieces, while retaining the ability potentially to define new models as well. The idea behind an `MCMCArgument` is to decompose the task of defining the value of a bond $b(\theta)$, where θ denotes the nodes that are part of the bond b, into writing

$$b(\theta) = \ell(A_0(\theta), \ldots, A_{N(b)-1}(\theta)),$$

and defining the `LogDensity` function ℓ and the `MCMCArgument` functions A_j. We have already seen the example

$$\ell_{\text{Gaussian}}(y, \mu, \sigma) = \sum_{i=0}^{n-1} \{-\log \sigma_i - (y_i - \mu_i)^2 / 2\sigma_i^2\}.$$

It remains to define `MCMCArgument` functions A_0, A_1, A_2 so that, in a manner of speaking, $A_0(\theta) = y$, $A_1(\theta) = \mu$, and $A_2(\theta) = \sigma$.

`MCMCArgument` is an interface with the single method `getArgument()`:

```
interface MCMCArgument {
    double[] getArgument (double[][] params);
}
```

The `getArgument()` method takes an array of arrays of reals as input; these are potential values of each of the nodes in the bond. The return value of the method is a (one-dimensional) array of outputs to be sent to the `LogDensity` object. Here is a partial list of useful `MCMCArgument` objects.

- `ConstantArgument`. This class ignores the values of the parameters and always returns a predetermined array of data, sometimes many copies of the same value, sometimes an array of different data points.
- `IdentityArgument`. This class returns one of the parameters unchanged.
- `GroupArgument`. This class is useful when, for example, many data points share the same mean. It "expands" a short vector of parameters into a larger vector. For example, if $y_i \sim N(\mu, \sigma^2)$ for $0 \leq i < n$, then since the μ and σ parameters have length one, two `GroupArgument` objects are needed, the first to create a vector of length n, each of whose elements is equal to the scalar μ, and another to do the

same for σ. Creating a `GroupArgument` is very natural in S: if `theta` is a parameter and `g` is a vector of group indices, the `GroupArgument` would calculate `theta[g]`.

- `FunctionalArgument`. The purpose of a `FunctionalArgument` is to run one or multiple parameters through an arbitrary function specified by the user, possibly after "expanding" them first as in `GroupArgument`. A simple example of a functional argument is the case where normal data have standard deviations proportional to their means: in this case a `FunctionalArgument` first expands a scale parameter to the length of the data, and then takes the absolute value of the product of this scale parameter and the means.

- `LinearModelArgument`. The class `LinearModelArgument` defines linear models, including those with both numeric and categorical variables. It also contains the capability for running the linear predictor through an arbitrary (link) function. These arguments can then function as the mean or the standard deviation in a normal linear model, or, if a different `LogDensity` is used, in a generalized linear model.

2.3.3 Other types of `MCMCBond`

A natural generalization of the `BasicMCMCBond` involves mixture distributions. Suppose that unknown parameters θ may satisfy one of several bond functions $b_i(\theta)$, and the probability associated with the ith bond, possibly also depending on unknown parameters, is $w_i(\theta)$. The unnormalized posterior distribution will then include the term $\sum_i w_i(\theta)b_i(\theta)$. This design describes the b_i using `BasicMCMCBond` objects, and the w_i using an additional `MCMCArgument`.

Another potentially useful construct is the `PowerBond`, in which a `BasicMCMCBond` is raised to a power other than one, so that the posterior distribution contains a term $b(\theta)^{h(\theta)}$. `PowerBond` objects can be used in importance sampling and simulated tempering (Geyer and Thompson, 1995).

2.4 Updates: steps in the MCMC algorithm

A versatile architecture for defining statistical models in MCMC software is not of much use unless there is hope that the software can generate or help the user generate algorithms that can sample from the posterior distribution reasonably efficiently. Algorithms in this design are defined to be arrays of objects implementing the `MCMCUpdate` interface. Classes that implement this interface include `MCMCParameter` itself, with the result that a default variable-at-a-time, random walk Metropolis algorithm can be constructed with no more effort than listing the nodes. In some cases this default algorithm will lead to unacceptably high autocorrelations. If so, other objects implementing `MCMCUpdate` can be added to the array of update steps to try to improve the mixing.

The `MCMCUpdate` interface consists of one critical method:

```
interface MCMCUpdate {
    public void update ();
}
```

A generic algorithm in this design is built by adding objects implementing `MCMCUpdate` to a collection, and the algorithm is then run for one iteration by looping through the objects in the collection, calling the `update()` method of each.

The YADAS implementation of the interface also includes the methods `String accepted()` and `void updateoutput()`, which generate output related to acceptance rates of the algorithm, but these are not critical to the design.

As described, the `MCMCUpdate` interface puts no restrictions on the algorithms that can be part of this design but also provides no assistance in creating useful algorithms. Specific classes that implement the interface are responsible for the diversity of useful algorithms.

2.4.1 `MCMCParameter` as an update

One class that implements `MCMCUpdate` is `MCMCParameter` itself. A parameter is an update in the sense that an often useful step in an MCMC algorithm is to update a single scalar parameter by proposing a (for example) Gaussian perturbation to it and accepting or rejecting using the Metropolis rule. The acceptance probability for the move is calculated by calling the `relevantBonds()` method for the parameter discussed in the `MCMCNode` section, and computing the sum of these bond values as a function of the current parameter value and the proposed perturbation. Since an `MCMCParameter` in general consists of multiple scalar parameters (components), the implementation of its `update()` method loops over these components, attempting a random walk Metropolis step for each, each with its own proposal standard deviation. Therefore, one can define an algorithm trivially, by listing the names of all the parameters. In YADAS, the initial definition of an `MCMCParameter` includes, along with an array of initial values, an array of nonnegative step sizes that are used as the standard deviations of proposal distributions.

The `MCMCParameter` class also has some useful subclasses that are identical in functionality except for their different `update()` methods. In particular, objects in the `MultiplicativeMCMCParameter` class have positive values and update themselves with Gaussian random walk proposals on the log scale, while `LogitMCMCParameter` objects have support on $(0, 1)$ and make proposals on the logit scale. `IntegerMCMCParameter` handles parameters whose possible values are a subset of the nonnegative integers: its `update()` method

proposes a new value using a Gaussian random variable rounded to the nearest integer. If these classes are used in an analysis, it is still possible to define an algorithm by listing all the nodes.

2.4.2 The `MultipleParameterUpdate` class

Individual updates to each scalar parameter can define an algorithm with which to start an analysis. These algorithms can perform poorly when the posterior distribution features high correlations between parameters. In some cases reparameterization solves this problem (see, for example, Gelfand and Sahu, 1999) but this requires major changes to the MCMC code. Instead, one can use Metropolis or Metropolis–Hastings updates in which multiple parameters and/or multiple components of individual parameters are updated simultaneously; see Graves, Speckman and Sun (2005). This approach takes advantage of the software architecture based on calculating posterior ratios. The `MultipleParameterUpdate` class facilitates defining these additional steps: in the notation of §2.3.2, the user defines how to sample from a family of distributions $T(\cdot|\theta)$, and also how to calculate $T(\theta'|\theta)$. The definition of a `MultipleParameterUpdate` is a set of `MCMCNode` objects and an object implementing the `Perturber` interface. The `Perturber` interface is defined as follows:

```
interface Perturber {
    void perturb (double[][] candarray, int whoseTurn);
    int numTurns ();
    double transitionDensityRatio ();
}
```

The `Perturber` object's methods define how to sample from and calculate one or several T_j's. The `perturb()` method accepts an array of arrays of current values θ of parameters, as well as an integer indicating which T_j is to be attempted, and changes the elements in the array of arrays according to a sample θ' from $T_j(\cdot|\theta)$. Note that depending on the programming language, it may not work to change the elements of the array in place and the method may have to return the array itself rather than `void`. The method `numTurns()` returns the number of distinct moves T_j that this `MultipleParameterUpdate` will attempt in a single iteration of the MCMC algorithm, and the method `transitionDensityRatio()` calculates $T_j(\theta|\theta')/T_j(\theta'|\theta)$. Examples of `Perturber` objects include adding a common constant to each component of a vector parameter and to the scalar parameter that serves as the mean of the vector parameter. It is also possible to modify similarly the scale of entire groups of parameters rather than their center. In regression problems where it is not possible to center the data, one may wish to add a constant to the intercept α while simultaneously subtracting a related constant from the slope β in order to keep $\alpha + \beta\bar{x}$ constant, where $\bar{x}$ is the average of the covariates. If an unknown parameter $\{p_i : 0 \le i < n\}$ satisfies $\sum_i p_i = 1$, its proposed updates must respect this constraint, and `SumToOnePerturber`, whose i'th step proposes a Gaussian perturbation to p_i on the logit

scale and rescales the remaining p_j accordingly, is one way to accomplish this.

2.4.3 Other `MCMCUpdate`'s

Another class of updates can be used when an updatable parameter has finite support. If this support is finite, then `FiniteUpdate` can handle this case using a Gibbs step, by computing the posterior ratio for each possible move of the parameter, and selecting the next value of parameter proportionally to these ratios. See §3.1 for an example.

Another update featured in YADAS supports reversible jump MCMC (Green, 1995). Reversible jump MCMC can be used when a parameter's support is best thought of as a union of two or more spaces. The `ReversibleJumpUpdate` class is used here: conceptually, this class requires a matrix of transition probabilities for proposing moves between the spaces, and a matrix of proposed moves for each ordered pair of spaces. The user also needs to supply transition density functions for these moves, so that use of this class is not as automatic as in other classes.

2.4.4 `UpdateTuner` and `TunableMCMCUpdate`

The interface `MCMCUpdate` can be extended to make update steps self-tuning, for example by permitting automatic tuning of proposal standard deviations. The current YADAS approach, described in Graves (2005), uses the class `UpdateTuner`, which operates on objects implementing the `TunableMCMCUpdate` interface. This approach features a trial stage that experiments with multiple step sizes and fits a logistic regression to acceptance probability as a function of log(step size), then utilizes this estimated logistic regression model to choose a step size corresponding to a target acceptance rate. This approach has been used successfully with `MCMCParameter`, `MultiplicativeMCMCParameter`, `LogitMCMCParameter` and `MultipleParameterUpdate` objects.

2.5 Output

In YADAS, MCMC output has been restricted to writing successive posterior samples to files for later analysis with other software. Here a potential output is anything that can be done at the close of some or all complete MCMC iterations. The definition for the interface `MCMCOutput` is

```
interface MCMCOutput {
    public void output ();
```

```
}
```

The most obvious example of an output is writing the latest posterior sample to a file; the class `MCMCParameter` implements `MCMCOutput` and does exactly that. Another possibility is a class that periodically creates trace plots of parameters. Also, a class implementing `MCMCOutput` could efficiently calculate mean and covariance matrix estimates, and these could be fed back to an `MCMCUpdate` in an adaptive algorithm.

2.6 Running

After all the parameters, bonds, and updates in an application are defined, the remainder of the code in an application is simple. The algorithm consists of looping over the array of `MCMCUpdate` objects, attempting each update in each iteration. Most of these updates in turn loop over the bonds in order to calculate ratios of values of the posterior distribution. After the last update of an iteration has been attempted, one sends the current values of the parameters to output files. For a complete YADAS example, see the Appendix.

3 Examples

This section discusses two examples that illustrate some of the strengths of the design. First, in 3.1 discusses estimating the prevalence of a binary characteristic in a population, where the prevalence varies across manufactured lots, and where the data include both random sampling and convenience sampling from the lots. This example illustrates how the design incorporates novel probability models and facilitates algorithms with discrete parameters. The second example concerns analyzing the results of auto races under a rich family of models for permutation data (§3.2), illustrating how the design is favorable for a sequence of analyses under related models, and also how algorithms can be improved once poor mixing is identified.

3.1 Random and convenience sampling from lots

Consider the estimation of the fraction of items in a finite population that have a certain feature; see Graves *et al* (2003). The items were manufactured in lots. To indicate that presence or absence of the feature is potentially related to lot membership, let p_i be the probability that an item manufactured in lot i contains the feature, and assume that the p_i's come from a Beta(a, b) distribution, where a and b are also to be estimated. Items were sampled from some of the lots and inspected to determine whether they contained the feature, but the sampling proceeded in two phases, and only the second phase generated random samples within the lots. The first "convenience" sample was nonrandom, and the selection mechanism may not have

been independent of the presence or absence of the feature. For this reason, the samples were analyzed using the *extended hypergeometric* distribution. Let N_i be the number of items in lot i, suppose that K_i of these items contain the feature, where K_i has a binomial distribution with parameters N_i and p_i, and suppose that the size of the ith convenience sample was n_i^C. Let θ be a parameter that measures the extent to which items with the feature are more likely to be sampled than items without ("bias"). Then the probability distribution for y_i^C, the number of items inspected in the convenience sample that contain the feature, is

$$f(y; N_i, n_i^C, K_i, \theta) = \frac{\left(\begin{array}{c} n_i^C \\ y \end{array} \right) \left(\begin{array}{c} N_i - n_i^C \\ K_i - y \end{array} \right) \exp(\theta y)}{\sum_{j=\max(0, n_i^C - N_i + K_i)}^{\min(n_i^C, K_i)} \left(\begin{array}{c} n_i^C \\ j \end{array} \right) \left(\begin{array}{c} N_i - n_i^C \\ K_i - j \end{array} \right) \exp(\theta j)} \ ,$$

for $y = \max(0, n_i^C - N_i + K_i)$, ..., $\min(n_i^C, K_i)$.

This reduces to the hypergeometric distribution when $\theta = 0$, and when $\theta > 0$, items with the feature are more likely to be sampled. A $N(0, 1)$ prior on θ allows for the possibility that the sampling was biased without assuming a direction for the bias. The extended hypergeometric distribution (Harkness, 1965) also arises in the power function for Fisher's exact test for 2×2 tables. The randomly sampled data, taken from what remains after the convenience samples, are then modeled using the hypergeometric distribution.

The parameters of most interest are the K_i's, because from them, one can derive the number of features in the uninspected part of the population. The potentially challenging aspects of this problem are the extended hypergeometric density and the discrete support of the main parameters of interest. Using this software design, however, this problem is not difficult. The `ExtendedHypergeometric` class created for this problem implements the `LogDensity` interface, so that it defines a `compute()` method that when called with vector arguments y, N, n, K, and θ, returns $\sum_i \log f(y_i; N_i, n_i, K_i, \theta_i)$. The randomly sampled data also highlight the `MCMCArgument` construct: the unknown number of features available to be sampled randomly is a function $K_i - y_i^C$ of the unknown parameters K_i, and these numbers are computed using a `FunctionalArgument` and then sent to the ordinary hypergeometric `LogDensity`: there is no need for a version suitable for two phases of sampling. Finally, the K_i are sampled using the `FiniteUpdate` class, which implements the `MCMCUpdate` interface, and samples from the complete conditional distributions of parameters whose support is $\{0, 1, \ldots, S-1\}$, where S is some positive integer. This design is conveniently set up to do this, since it is based on computing terms like $r_i = f(i)/f(s_0)$, where $f(\cdot)$ denotes the unnormalized posterior evaluated at the value of the discrete parameter and where s_0 is the current value. Sampling the new value of the discrete parameter with probabilities proportional to the r_i is then exactly a Gibbs step for this parameter. Another option for sampling discrete parameters is the `IntegerMCMCParameter` class, a subclass of `MCMCParameter`, which implements a Metropolis step where the proposed value of the discrete parameter is a discretized

Gaussian step from the current value and doesn't compute all possible moves, some of which may be very unlikely.

3.2 Auto racing results

A second example that illustrates families of analyses using of a single new `LogDensity` object involves analysis of auto racing results. Graves, Reese, and Fitzgerald (2003) present a family of models for permutations that depend on ability parameters for each driver; the individual models differ in the structure of the ability parameters and estimate interactions between drivers and various factors. The basic model is as follows. Suppose n drivers participate in a race. Let the ability of the ith driver be denoted by θ_i, where the θ_i's have an independent mean zero normal distribution with unknown variance. The probabilistic model (the "attrition model") determines finishing order conditional on ability parameters by choosing the last place finisher with probability proportional to $\lambda_i = \exp(-\theta_i)$, choosing the second-to-last finisher among the remaining drivers with probabilities still proportional to λ_i, and continuing until two drivers i_1 and i_2 have been determined to finish first and second; driver i_1 wins with probability $\lambda_{i_2}/(\lambda_{i_1} + \lambda_{i_2})$. To analyze many races simultaneously, assume that driver i has ability θ_{ij} in race j, and let $\lambda_{ij} = \exp(-\theta_{ij})$. Let π_j be defined so that $\pi_j(k) = i$ means that driver i finished in kth place in race j; then the likelihood function is

$$P(\pi|\lambda) = \prod_{j=1}^{J} \prod_{i=1}^{I_j} \lambda_{\pi_j(i),j} \left(\sum_{k=1}^{i} \lambda_{\pi_j(k),j} \right)^{-1}. \tag{1}$$

Interesting ways of parameterizing θ_{ij} include $\theta_{ij} = \theta_i$ (driver abilities are constant over all races) and assuming that driver abilities are changing linearly in time. One may also study the extent to which some tracks are more predictable than others (i.e., $\theta_{ij} = \theta_i \phi_{T(j)}$, with large values of the positive parameter ϕ_t implying that results at track t are highly predictable). We also studied the possible existence of track-driver interactions: each track-driver combination (i, t) leads to a different parameter Ω_{it}, with the Ω_{it}'s distributed around an overall driver ability θ_i. One may also propose regression relationships based on track properties like speed, length, and amount of banking.

Using this software design, one studies all these models using a single object implementing the `LogDensity` interface called `AttritionLikelihood`, which stores a vector of race numbers and a vector of finish positions, and takes two arguments corresponding to a vector of θ's parameterizing driver abilities (these can depend on properties of the race) and a vector of ϕ's parameterizing the predictability of the races (most often, these ϕ's are identically one). `AttritionLikelihood` then computes the log of the probability (1), conditional on the unknown parameters. The θ and ϕ vectors can easily be constructed using objects implementing the `MCMCArgument` interface, typically `GroupArgument` objects.

The auto racing MCMC algorithms have also benefited from some special update steps. For example, suppose that the prior distribution for the θ's is independent $N(0, \gamma^2)$, and γ has a gamma hyperprior distribution. Then γ and the sample standard deviation of the θ's are highly correlated, and the parameter γ can mix slowly. An algorithm with better mixing with respect to γ can be obtained by adding a `MultipleParameterUpdate` where the proposed move is $T(\theta_1, \ldots, \theta_N, \gamma) = (\bar{\theta}+c(\theta_1-\bar{\theta}), \ldots, \bar{\theta}+c(\theta_N-\bar{\theta}), c\gamma)$, where $\bar{\theta} = N^{-1}\sum_1^N \theta_j$ and where c is chosen lognormally. Augmenting algorithms with moves like these is easy, as it amounts to composing appropriate `Perturber` functions.

3.3 Other possibilities

This design has been utilized experimentally in a wide variety of problems and in different algorithms. These analyses and algorithms have not all been complete successes from a scientific or algorithmic perspective, but they illustrate the potential of the design. Examples include reliability of complex systems with different data models for each component (Johnson *et al*, 2003; Graves and Hamada, 2005); modeling of symptom counts using a fixed seasonal shape but hierarchical models for the baseline, amplitude, and peak location (Graves and Picard, 2002); flexible models for edge presence in random graphs; software reliability with a hierarchical model on parameters with mixture distributions; simulated tempering as in Geyer and Thompson (1995); `LogDensity` and `MCMCArgument` objects that use sufficient statistics for several linear regression models; Gaussian spatial process models; discretely supported parameters sampled using continuous versions; arguments defined using kernel smoothing; path sampling (Gelman and Meng, 1998); and approximations of parameter-expanded data augmentation (Meng and van Dyk, 1999; Liu and Wu, 1999)) and alternating subspace-spanning resampling (Liu, 2003).

4 Related Work

Several general–purpose MCMC tools are available. First, the free, versatile, and usable package BUGS (Spiegelhalter *et al*, 1996) deserves special notice. This package has an intuitive language (as well as a graphical user interface) for expressing models, and does not require (or allow) the user to specify an algorithm with which to attack a problem. In the author's opinion, if a problem can be handled by BUGS, then as of today one should normally use BUGS to solve it. An open source implementation called OpenBugs (see `http://mathstat.helsinki.fi/openbugs/`) is intended to become the standard version. JAGS (Plummer, 2003) is an open-source emulation of BUGS written intended to be an extensible experimentation platform.

MLwiN (Rasbash *et al*, 2000) is another package with many features including a graphical user interface and graphical diagnostics. This package supports classical analyses as well as MCMC. Most of their examples

are hierarchical generalized linear models, admittedly a large and useful class of problems, but it does not appear to be straightforward for users to extend the capabilities of the package. MLwiN is not free.

Bassist (Toivonen *et al*, 1999) is another good package; unfortunately, it appears that it is no longer being supported. It converts code written using a model description language into C++ code. It has good extensibility since it can use C++ functions to define model parameters, but it is limited in its capability for generating different sorts of MCMC algorithms.

HYDRA (Warnes, 2002) is also written in Java and distributed under the open-source model. It does not seem to have any limits to its extensibility, but neither does it provide classes that make it easy to specify new models (one has to define the posterior distribution from scratch) or to define new MCMC algorithms, although several special algorithms are included.

Flexible Bayesian Modeling (FBM) is a set of UNIX based tools for many sorts of Bayesian regression models. See Neal (2003).

Many excellent R packages (R Development Core Team, 2005) include special–purpose Bayesian analysis code. Several packages aspire to greater generality. Kerman (2006)'s Umacs is promising: the user needs to define most of the details of each parameter's updating mechanism which can be either a Gibbs or a vector or scalar Metropolis step, and Umacs runs the sampler, monitors acceptance rates and thereby tunes proposal kernels. Also, Geyer (2005) provides the `mcmc` package, which features the beginnings of a general MCMC framework, accepting arbitrary unnormalized densities and arbitrary proposal covariance matrix. Martin and Quinn (2005) features MCMC algorithms for many models and includes a general random walk Metropolis function.

5 Discussion

The design ideas discussed in this paper have been useful in many analyses. Areas for future work include the development of an error handling interface. Implementation of these design ideas in other languages could expand their use. Also, a long-term goal of the YADAS project is to have software that will automatically generate acceptable algorithms for as many problems as possible, so that the user's responsibility is only to define the model. The design's decoupling of the algorithm from the model is an important step toward this goal. The goal is remote at this point, so the software supports the user in defining his or her own algorithms and experimenting with them: the design should support an environment for research into new methods for improving mixing in MCMC.

Appendix: Example of YADAS code

YADAS is a collection of Java classes. The YADAS source code, a tutorial, examples, and supporting documents are available for download at `yadas.lanl.gov`. Below is code for a simple YADAS analysis (estimation of the group means and variance components in a one-way analysis of variance). As mentioned earlier, `ArgumentMaker` is the YADAS name for the concept `MCMCArgument` discussed in this paper. This code is contained in a file called `OneWayAnova.java` and compiled into a class called `OneWayAnova.class`. The code's five stages are described below.

```
import gov.lanl.yadas.*;
public class OneWayAnova {
  public static void main (String[] args) {
    MCMCAnalysis mcmc = new MCMCAnalysis ("~/analyses/onewayexample/");
    // 1. Assemble input sources
    DataFrame d, d2; ScalarFrame d0;
    mcmc.addInput (d = new DataFrame ("data.dat"));
    mcmc.addInput (d2 = new DataFrame ("mu.dat"));
    mcmc.addInput (d0 = new ScalarFrame ("scalars.dat"));
    // 2. Define unknown parameters
    MCMCParameter mu, theta, sigma, delta;
    mcmc.addNode (mu = new MCMCParameter (d2.r("mu"), d2.r("m_ss"), "mu"));
    mcmc.addNode (theta = new MCMCParameter (d0.r("theta"), d0.r("t_ss"), "theta"));
    mcmc.addNode (sigma = new MCMCParameter (d0.r("sigma"), d0.r("s_ss"), "sigma"));
    mcmc.addNode (delta = new MCMCParameter (d0.r("delta"), d0.r("d_ss"), "delta"));
    // 3. Define model
    mcmc.addBond ( new BasicMCMCBond
        ( new MCMCParameter[] { mu, sigma },
          new ArgumentMaker[]
              { new ConstantArgument (d.r("y")),
                new GroupArgument (0, d.i("group")),
                new GroupArgument (1, d.i(0)) },
          new Gaussian () ));
    mcmc.addBond ( new BasicMCMCBond
        ( new MCMCParameter[] { mu, theta, delta },
          new ArgumentMaker[]
            { new IdentityArgument (0),
              new GroupArgument (1, d2.i(0)),
              new GroupArgument (2, d2.i(0)) },
          new Gaussian () ));
    mcmc.addBond ( new BasicMCMCBond
        ( new MCMCParameter[] { sigma },
          new ArgumentMaker[]
            { new IdentityArgument (0),
              new ConstantArgument (d0.r("asigma")),
              new ConstantArgument (d0.r("bsigma")) },
          new Gamma () ));
    mcmc.addBond ( new BasicMCMCBond
        ( new MCMCParameter[] { delta },
          new ArgumentMaker[]
            { new IdentityArgument (0),
              new ConstantArgument (d0.r("adelta")),
              new ConstantArgument (d0.r("bdelta")) },
          new Gamma () ));
    // 4. Define algorithm
    mcmc.addUpdates ( new MCMCUpdate[] { mu, theta, sigma, delta } );
```

```
    // 5. Define output
    mcmc.addOutputs ( new MCMCOutput[] { mu, theta, sigma, delta } );
    mcmc.iterate (10000);
    mcmc.finish ();
    }
}
```

The first stage of the code defines inputs to the analysis: two `DataFrame` objects, one (`d`) containing the N data points and group labels, and one (`d2`) containing information about the K group means, and another input file `d0` containing scalar inputs. Second, define four unknown parameters: the multivariate parameter μ and the three scalar parameters θ, σ, and δ, assigning their initial values, step sizes, and output file names. The third step is to define the unnormalized posterior distribution. The four successive bond definitions mean, respectively,

1. for $i = 0, \ldots, N-1, y_i \sim N(\mu_{g_i}, \sigma^2)$, where g is an array of group labels;
2. for $k = 0, \ldots, K-1, \mu_k \sim N(\theta, \delta^2)$;
3. $\sigma \sim \Gamma(a_\sigma, b_\sigma)$;
4. $\delta \sim \Gamma(a_\delta, b_\sigma)$.

Implicitly, θ has a flat prior. The fourth stage of the code defines the algorithm: in this case, it is the default variable-at-a-time Metropolis algorithm. The fifth stage defines the default output mechanism: each parameter will generate a text file of MCMC samples.

Acknowledgments

The author thanks JCGS editor Luke Tierney. The author also thanks Mike Hamada for his constant support with the YADAS project and this paper.

References

Berger, J.O. (2000), "Bayesian Analysis: A Look at Today and Thoughts of Tomorrow," *Journal of the American Statistical Assocation* 95:1269-1276.

Chambers, J.M. (1998). *Programming with Data: A Guide to the S Language.* Springer-Verlag, New York.

Chib, S. and Greenberg, E. (1995), "Understanding the Metropolis–Hastings Algorithm," *The American Statistician* 49:327-335.

Eckel, B. (1998), *Thinking in Java*. Upper Saddle River, NJ: Prentice Hall.

Gelfand, A.E. and Sahu, S.K. (1999), "Identifiability, Improper Priors, and Gibbs Sampling for Generalized Linear Models," *Journal ofthe American Statistical Assocation* 94:247-253.

Gelman, A. and Meng, X.L. (1998). "Simulating Normalizing Constants: from Importance Sampling to Bridge Sampling to Path Sampling," *Statistical Science* 13:163-185.

Geyer, C.J. (2005), "mcmc: Markov Chain Monte Carlo." R package version 0.5. `http://www.stat.umn.edu/geyer/mcmc/`.

Geyer, C.J. and Thompson, E.A. (1995). "Annealing Markov chain Monte Carlo with applications to ancestral inference." *Journal of the American Statistical Assocation* 90:909-920.

Graves, T.L. (2003), "An Introduction to YADAS," `yadas.lanl.gov`.

Graves, T.L. (2005), "Automatic Step Size Selection in Random Walk Metropolis Algorithms," Los Alamos National Laboratory Report LA-UR-05-2359.

Graves, T.L. and Hamada, M.S. (2005), "Bayesian Methods for Assessing System Reliability: Models and Computation," chapter in Modern Statistical and Mathematical Methods in Reliability, World Scientific Publishing Company.

Graves, T.L., Hamada, M.S., Booker, J.M., DeCroix, M., Bowyer, C., and Chilcoat, K. (2003), "Inference for a Population Proportion Using Stratified Data Arising From Both Convenience and Random Samples," Los Alamos National Laboratory Report LA-UR-03-8396.

Graves, T.L. and Picard, R.R. (2002), "Seasonal Evolution of Influenza-Related Mortality," Los Alamos National Laboratory Report LA-UR-03-1237.

Graves, Reese, C.S., and Fitzgerald, M. (2003), "Hierarchical Models for Permutations: Analysis of Auto Racing Results," *Journal of the American Statistical Association*, 98:282-291.

Graves, T.L., Speckman, P., and Sun, D. (2005), "Characterizing and Eradicating Autocorrelation in MCMC Algorithms for Linear Models" Los Alamos National Laboratory Report LA-UR 04-0486.

Green, P.J. (1995), "Reversible Jump MCMC Computation and Bayesian Model Determination," *Biometrika* 82:711-732.

Harkness, W.L. (1965), "Properties of the Extended Hypergeometric Distribution," *Annals of Mathematical Statistics* 36:938-945.

Hastings, W.K. (1970), "Monte Carlo Sampling Methods Using Markov Chains and Their Applications," *Biometrika* 57:97-109.

Johnson, V.E., Graves, T.L., Hamada, M.S., and Reese, C.S. (2003), "A Hierarchical Model for Estimating the Reliability of Complex Systems (with Discussion)," Bayesian Statistics 7, Oxford University Press, Eds. J.M. Bernardo, M.J. Bayarri, J. Berger, A.P. Dawid, D. Heckerman, A.F.M. Smith, and M. West, 199-213 (2003). *Bayesian Statistics 7*, Oxford University Press.

Kerman, J. (2006), Umacs: A Universal Markov Chain Sampler. Technical Report, Columbia University, New York.

Liu, C. (2003). "Alternating Subspace-Spanning Resampling to Accelerate Markov Chain Monte Carlo Simulation", *Journal of the American Statistical Association* 98:110-117.

Liu, J.S. and Wu, Y.N. (1999), "Parameter Expansion for Data Augmentation," *Journal of the American Statistical Association* 94:1264-1274.

Martin, A.D. and Quinn, K.M. (2005), "MCMCpack: Markov chain Monte Carlo (MCMC) Package. R package version 0.6-5. `http://mcmcpack.wustl.edu`.

Meng, X.L. and van Dyk, D.A. (1999), "Seeking Efficient Data Augmentation Schemes via Conditional and Marginal Augmentation," *Biometrika* 86:301-320.

Metropolis, N., Rosenbluth, A.W., Rosenbluth, M.N., Teller, A.H., and Teller, E. (1953), "Equations of State Calculations by Fast Computing Machines," *Journal of Chemical Physics* 21:1087-1092.

Neal, R. (2003), "Software for Flexible Bayesian Modeling and Markov Chain Sampling"; see `http://www.cs.toronto.edu/~radford/fbm.software.html`.

Plummer, M. (2003), "JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling," *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003)*, Vienna, Austria, F. Leisch and A. Zeileis, eds.

R Development Core Team (2005), "R: A Language and Environment for Statistical Computing," R Foundation for Statistical Computing, Vienna, Austria, `www.R-project.org`.

Rasbash, J., Browne, W., Goldstein, H., Yang, M., Plewis, I., Healy, M., Woodhouse, G., Draper, D., Langford, I., and Lewis, T. (2000), *A User's Guide to MLwiN (Second Edition)*, London, Institute of Education.

Spiegelhalter, D.J., Thomas, A., Best, N.G., and Gilks, W.R. (1996), *BUGS: Bayesian inference Using Gibbs Sampling, Version 0.5 (version ii)*; see `http://www.mrc-bsu.cam.ac.uk/bugs/welcome.shtml`.

Tanner, M. and Wong, W. (1987) "The calculation of posterior distributions by data augmentation (with discussion)," *Journal of the American Statistical Association* 81:528-550.

Tierney, L. (1994), "Markov Chains for Exploring Posterior Distributions," *Annals of Statistics* 22:1701-1762.

Toivonen, H., Mannila, H., Seppanen, J., and Vasko, K. (1999), *Bassist User's Guide For Version 0.8.3*; see `http://www.cs.helsinki.fi/research/fdk/bassist/`.

Warnes, G.R. (2002), "HYDRA: a Java Library for Markov Chain Monte Carlo", *Journal of Statistical Software*, Volume 7 Issue 04, 03/10/02.